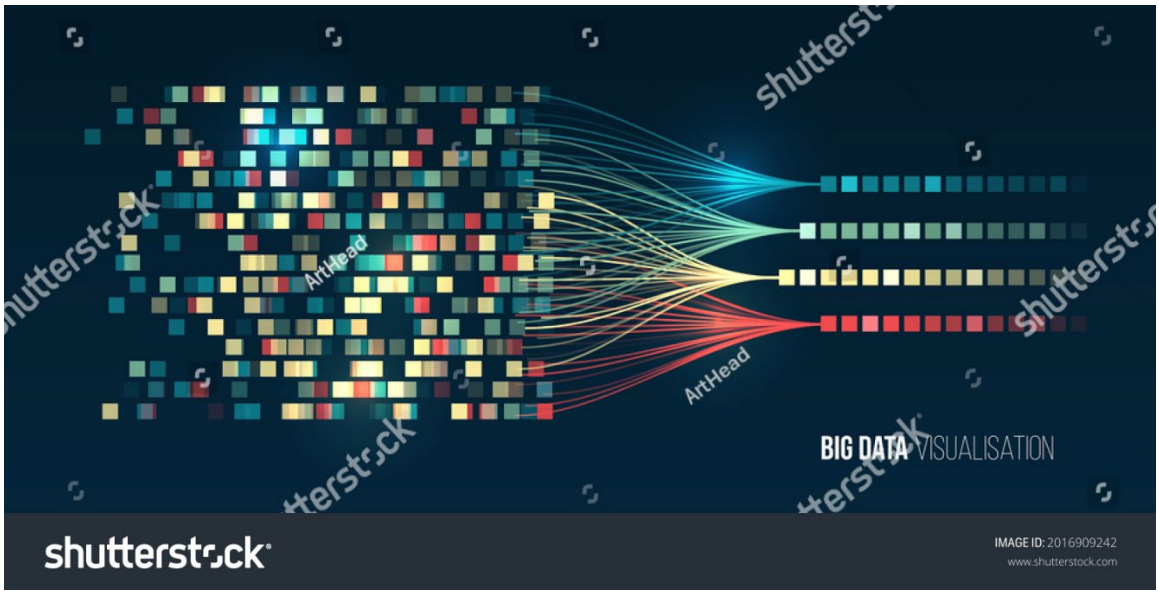


Implementation of AVL Trees and Stack for word processing

ASSIGNMENT 1

Sami Sheikh | CSCI203 | 19/8/2023

Solution Description

Function main:

Instantiate the AVLTree class using 'tree'

Read the filename from console

Open file with the user-given filename

If file is not opened successfully, report an error and exit the program.

While there are words to read from the file:

Read a word from the file into the 'word' variable

Use stripPunctuation(word) to remove punctuation

toLowerCase(word) to convert the complete word to lowercase

Search the 'tree' for 'word'

If the word exists in 'tree':

Increment the word count

Else:

Create a new node in the AVL tree with the word and insert into 'tree'

End while

Close the file

Display the top 10 words using the tree's printTop10()

Display the last 10 words using the tree's printLast10()

Display all unique words using the tree's printUniqueWords()

End function

Complexity analysis

For the program, the complexity comes from four main computations, **Reading and processing words**, **Printing Top 10 Words, Last 10 Words** and **Displaying Unique Words**.

In this analysis, "m" is the number of words in the file and "n" is the number of unique words present in the file.

READING AND PROCESSING WORDS

For every word that is read from the file, insertion and search is being utilized in the AVL tree which takes a O(log n) time complexity on average (where n is the number of unique words present in the file.)

If there are a total of 'm' words in the file, the complexity becomes O(m log n).

The O(log n) complexity of AVL tree insertion is due to its inherent property as a balanced binary search tree.. As a balanced binary search tree, the height of the AVL tree is always maintained within the logarithmic bounds of its number of nodes. Which ensures that any operation like searching, insertion or deletion will take at most O(log n) time. The rotational feature of the tree ensures its balancing which allows the tree's height to retain its logarithmic property relative to its size. Moreover, rotations that are involved in AVL trees for balancing are O(1)

PRINTING TOP 10 WORDS

In this instance, I used the "MaxHeap" data structure and algorithm where insertion is O(log k) where 'k' is the heap size (in this case, 10.) Moreover, since the inOrder traversal of the AVL tree is O(n), the overall complexity in building the heap will be O (n log k)

However, since k = 10 constant, it is simplified to O(n)

PRINTING BOTTOM 10 WORDS

The logic is similar to the earlier "Printing top 10" algorithm, however, the values here are placed in an array with rotations. The overall complexity of this is O(n).

Furthermore, regarding the top and bottom 10 words algorithm. Heap operations are typically done in O(log k) time complexity where k = heap size. However, since we are

only concerned with 10 words, k = 10. The use of AVL trees and their in-order traversals along with parallel insertion and deletion of the heap, introduces an O(n) complexity.

The AVL Tree allows for efficient access and processing of data, and the heap allows for effective structuring and management of the top and bottom words.

DISPLAYING UNIQUE WORDS

In this instance, it is the traversal of the AVL tree, which will take a simple O(n) complexity.

ALGORITHM CONCLUSION

Overall, the time complexity of the solution I've provided is:

$$O(m \log n)$$

Data Structures used and reasons

- **AVL Trees (word storage and counting)**
    - I have used the AVL tree because as a balanced binary tree, it allows for quick insertion, deletions and searching that are logarithmic in complexity. Moreover, it also allows for quick traversal and processing of stored data read from the input file along with insertion and count updates.

- **Max Heap (Tracking top words by count) and Min Heap (Tracking bottom words by count)**

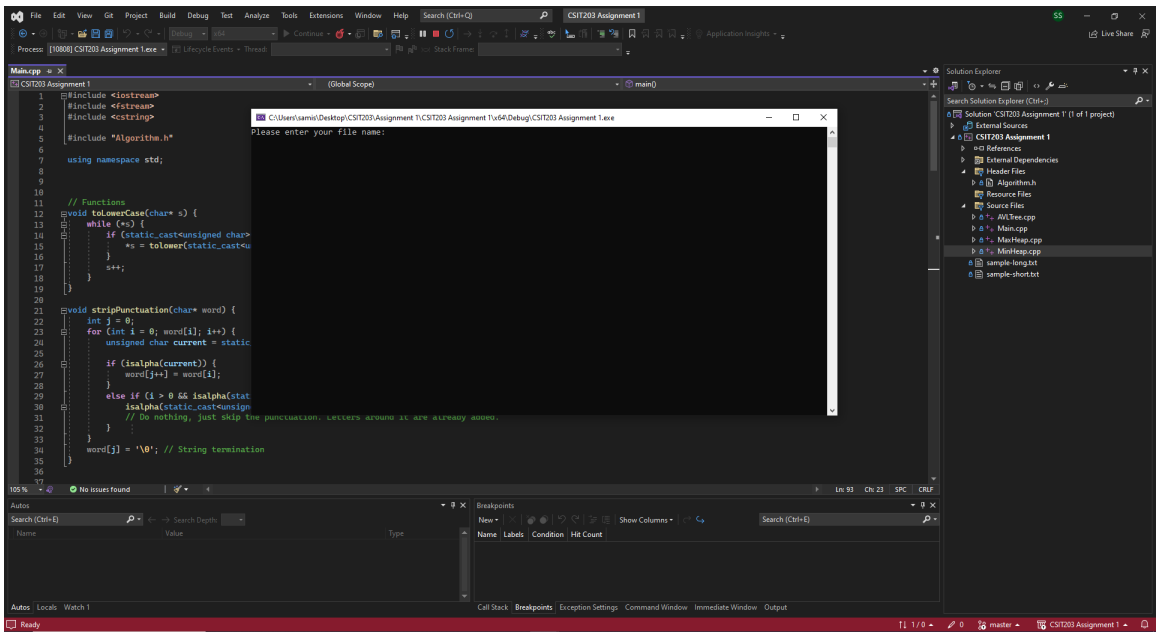
- I have used heap in this instance because of its efficiency in the insertion and removal of least frequent words. It allowed for quick insertions from the AVL Tree along with easy retrievals.

Compilation Snapshot and Execution of “sample-long.txt”

Microsoft Visual Studio Debug Console

```
Please enter your file name:
sample-long.txt
File: sample-long.txt has been opened successfully!

Top 10 Words:
the: 131
about: 7
acquaintance: 3
above: 2
accept: 2
abuse: 1
accomplished: 1
account: 1
acknowledged: 1
acquaintances: 1

Last 10 Words:
would: 10
wwwgutenbergorg: 1
year: 3
years: 2
yet: 1
you: 74
young: 7
youngest: 2
your: 12
yourself: 1

Unique Words:
agreed
addressed
acknowledged
abuse
accomplished
account
acquaintances
act
added
affect
admitted
admiration
adjusting
admire
advice
afterwards
afraid
agree
almost
air
ah
altogethermr
already
altogether
amends
amiable
amusement
amongst
angry
ankle
anywhere
anyone
arrival
arrived
```

```
third
throw
those
thought
threeandtwenty
tide
tiresome
times
title
towards
tomorrow
tolerable
toward
trimming
under
turned
tumult
turning
unaffected
uncertain
uncommonly
unreserved
universally
understanding
understand
unlucky
unless
unworthy
upper
various
using
venture
vexing
vexed
visiting
violent
village
waited
walking
wasting
wayswith
week
whatsoever
whichever
whom
wifes
window
wwwgutenbergorg
within
withdrew
without
wonderfully
wore
worth
yet
yourself

C:\Users\samis\Desktop\CSIT203\Assignment 1\CSIT203 Assignment 1\x64\Debug\CSIT203 Assignment 1.exe (process 15496) exited with code 0.
Press any key to close this window . . .
```



Conclusion

In this assignment, I have proposed the algorithm solution by using AVL Trees and Heaps. The use of both algorithms helped in achieving efficient insertion, counting and retrieval operations based on the scope and task presented in this assignment.

The AVL Tree, as a self-balanced tree assisted in insertions, deletions and searches that was carried out in logarithmic time, which gave the most performance advantage over regular binary search trees. Regular binary search trees would not have been as efficient with larger datasets without strong degradation in performance.

The use of heaps has also allowed me to store counts efficiently without having to sort through the entire AVL tree dataset. It allowed for quick retrieval and storage of the top and bottom 10 words. Considering the constant k = 10, it provided an O(n) complexity which is the best.

Moreover, the challenges that I've encountered through this assignment were in efficiently handling words as some words in the sample-long.txt file were not in regular ASCII code, where I had to utilize casting to ensure that proper ASCII was followed during toLowerCase() and stripPunctuation() functions. Moreover, the processing for converting the words into the AVL Tree, while also building one in C++ with manual memory handling was a good learning experience.

Further adjustments and optimizations to the code would be to analyze entire phrases and sentences rather than single words and/or implement other data structures such as hash maps.

In conclusion, this assignment pushed my coding limits to the edge, along with my understanding of data structures and algorithms. It also made me further understand the importance of selecting the right data structure based on the task given and how I can further optimize algorithms to handle data structures more effectively.